AUEB M.Sc. in Data Science (part-time)

Deep Learning

Homework 2

Course instructor: Prof. **P. Malakasiotis**

Student: **Spiros Politis** (P3351814)

Table of contents

Table of contents 1

Purpose and scope 2

Data Ingestion 2

Experimental setup 3

Experimental strategy 3

Setup 3

Optimizer, loss function and metrics 6

Deep neural architectures 7

Convolutional Neural Net (CNN) architecture 7

Transfer learning 10

Choosing the right architecture 10

DenseNet with pre-trained images on ImageNet 10

Ensemble learning 12

Data augmentation 13

Experimental results 14

Evaluation metric (ordered by test_auc) 14

Prediction metrics (ordered by kappa) 15

Conclusions 16

Purpose and scope

The purpose of this assignment is to explore different *Deep Learning* architectures, both custom-build and such that exploit *transfer learning* (e.g. *ResNet*, *DenseNet*), for the task of classifying X-rays between two classes (*normal*, *abnormal*). The training set provided is *Stanford's MURA v1.1*. A comprehensive description of the dataset is provided at [1].

Data Ingestion

A rather non-trivial problem ingesting data, when dealing with large datasets such as *MURA v1.1*, stems from the size of the dataset itself. Initial efforts to load the dataset and split into *train / validation / test* sets by using recursive directory access and combining retrieved images with class labels proved futile.

Fortunately, *TensorFlow* provides users with the mechanics to stream entire datasets as *Python* generators by using methods of class "*ImageDataGenerator*", from the "*tf.keras.preprocessing.image*" package, specifically method "*flow_from_dataframe*". As a bonus, the class provides users with the ability to automatically generate augmentations of the dataset (e.g. random image rotation, random image flipping etc.).

To achieve streaming of the dataset, therefore, we implemented class "*Ingest*", which takes care of loading, splitting, optionally augmenting and streaming the data to the model. We took a subset (*10%*) of the train data to be used as the *validation* set. We also made sure to retain the distribution of the dataset when splitting by performing stratified test / split on the test data. Further to stratifying the dataset, it is worth noting that we also included the relative weights of each class (*0: 0.83904058, 1: 1.23737487*) as input to the models, at training time.

[1] "MURA Dataset: Towards Radiologist-Level Abnormality ...." https://stanfordmlgroup.github.io/competitions/mura/. Accessed 28 Jun. 2020.

Since input data is images, it is worth mentioning that the input shape of images was chosen at (*224*, *224*) with *3* channels, in "*channels_last*" format. Choosing the input image dimensions is a balance between performance (training speed, avoiding OOM errors etc.) and classification accuracy. Shape (*224*, *224, 3*) was initially chosen because of requirements imposed by the *ResNet* architecture, when the default classification head is chosen as output and we adhered to this shape thereafter.

Note: a major shortcoming of "*ImageDataGenerator*" in *TensorFlow* is that it does not take image dimensions ratio into account, when performing resizing. In effect, images produced by the generator are distorted. We researched the issue and the only solution was found in the "monkey-patch" solution mentioned in [2]. Unfortunately, a significant amount of time was lost until we were able to diagnose, validate and correct the problem.

Experimental setup

Experimental strategy

The strategy to achieve the task at hand can be broken down in two approaches:

1. Classification of *normal* and *abnormal* cases on the entire dataset and
2. Classification of *normal* and *abnormal* cases per X-ray type (*XR_HAND*, *XR_WRIST* etc.)

Setup

In order to be able to execute experiments effectively, we had to be able to devise a mechanism by which experiments can be run consecutively and relatively without attention, because of the high resources required for executing each experiment and the

[2] "Keras center and random crop support for ... - gists · GitHub." 14 Jun. 2019, https://gist.github.com/rstml/bbd491287efc24133b90d4f7f3663905. Accessed 2 Jul. 2020.

consecutive long-running times such executions require. Therefore, we created the Python class "*Experiment*" which takes care of the problem by:

- Allowing end-users to define the Deep Learning architecture of the experiment to be executed (e.g. *CNN*, *DenseNet* etc.)
- Allowing for the parameterization of the experiment by enabling end-users to define dataset ingestion parameters (*batch size*, *validation split* etc.)
- Allowing for the parameterization of the *Deep Learning* architecture itself by enabling users to define parameters such as *optimizer*, *loss function*, *metrics* to be used etc.
- Providing the means to report on metrics per epoch by saving experimental runs output to the directory "*model/logs*".
- Providing the means to retrieve the best model checkpoint, as provided by the *early-stopping* mechanism, by saving the model to directory "*model/checkpoints*", for later reuse (model evaluation and inference).

In order to be able to discriminate between experimental runs output, we had to devise a file naming codification convention / scheme. Therefore, each output file name consists if the following (baseline) parts:

- <*model_architecture*>_: represents the *Deep Learning* architecture for which the experiment was run (e.g. "CNN", "DenseNet" etc.).
- <*da = data augmentation*>_<true / false value>_: whether data augmentation has been considered, encoded by the string "*da*" and followed by the respective value (0, 1).
- <*ts = target size*>_<tuple>_: the image input shape (width, height), encoded by the string "*ts*" and followed by its respective value.
- <*trainable*>_<true / false value>_: whether all models layers, when using weights from a pre-trained model, are trainable, encoded by the string "*t*" and followed by its respective value (0, 1).
- <*s*>_<true / false value>_: whether data has been shuffled, encoded by the string "*s*" and followed by its respective value (0, 1).

- <*bs* = *batch size*>_<numeric value>_: the batch size, represented as "*bs*" and followed by its respective value (e.g. 16, 32, etc.).
- <*class*>_: the experiment target class (e.g. "*all_classes*", "XR_ELBOW", "XR_WRIST" etc.).

An example of such an encoded file name is "*DenseNet169_da_0_ts_224_224_t_0_s_1_bs_16_XR_HUMERUS_binary_training*".

Further to this encoding, we had to devise a similar encoding scheme for capturing the parameters of the *CNN*:

- <*cl* = *convolutional layers length*>_: represents the number of convolutional layers of the *CNN*.
- <*kg* = *kernel growth*>_: indicates the *kernel* size increase or decrease scheme, as described in Convolutional Neural Net (CNN) architecture.
- <*fg* = *filter growth*>_: indicates the *filter* size increase or decrease scheme, as described in Convolutional Neural Net (CNN) architecture.
- <*lp* = *local pooling*>_: indicates whether a local pooling layer (*max* or *avg*) is used.
- <*gp* = *global pooling*>_: indicates whether a global pooling layer (*max* or *avg*) is used.
- <*sd* = *spatial dropout*>_: indicates whether a spatial dropout is used and its rate (e.g. *05* for *0.5*)..
- <*fcd* = *spatial dropout*>_: indicates whether dropout at the fully connected layer is used and its rate (e.g. *05* for *0.5*).

An example of such an encoded file name is "*CNN_da_0_ts_224_224_t_0_s_1_bs_8_cl_3_lp_max_gp_max_sd_05_fcd_05_all_classes_binary*".

The implementation of reporting for this method was done in *Python* class "", which performs all appropriate runs for producing results.

Optimizer, loss function and metrics

Considering the task at hand and considering the *MURA* contest, we realize that the appropriate measure to be used is *Cohen's Kappa* metric. However, we failed to apply it as a measure of conditioning during training time, perhaps due to the way of loading our training data. Nevertheless, we were able to evaluate the trained models on *Cohen's Kappa* during the model evaluation stage.

Considering that we are dealing with a problem of classification with unbalanced classes, we chose *ROC-AUC* as the measure to use because the curve balances the class sizes, as it will only actually select models that achieve false positive and true positive rates that are significantly above random chance, which is not guaranteed for accuracy. *AUC* measures how *true positive rate* (*recall*) and *false positive rate* trade off. More importantly, *AUC* is the evaluation of the classifier as the threshold varies over all possible values. It is a broader metric, testing the quality of the internal value that the classifier generates and then compares to a threshold. It is easy to achieve *99%* accuracy on a data set where *99%* of examples are in the same class. Thus, a perfect predictor gives an *ROC-AUC* score of *1.0* while a predictor which makes random guesses has an *ROC-AUC* score of *0.5*.

Since we are loading our target classes as *categoricals,* loss is computed based on the *categorical cross-entropy* loss function.

For our optimizer, we chose *Adam* since it has been demonstrated that it is the best for the task at hand.

During the model evaluation stage, we acquired the *loss*, *accuracy* and *AUC metrics*, measured on the model's performance on the test set. Furthermore, we measured each model's predictive power by acquiring their respective *accuracy* (for completeness, this measure is not very informative for our task, as mentioned previously)*, precision, recall, F1-score and Cohen Kappa* metrics*.*

Deep neural architectures

Convolutional Neural Net (CNN) architecture

Convolutional layers apply learned filters to input images in order to create feature maps that summarize the presence of input features. For our *CNN* architecture, we experimented with a variety of stacked *convolutional layers* as well as experimented with the following modes of building the progression of kernel and filter sizes:

- A *funnel-out* mode, in which *kernel* size **increases** incrementally with each layer as a progression *((i * 2) + 1, (i * 2) + 1),* e.g. *(3, 3), (5, 5), (7, 7), ...* and *filter* size **increases** as a progression *np.power(2, i + 4),* e.g. *32, 64, 128, ...,* where *i* is the current convolutional layer index.
- A "*funnel_in*" mode, in which *kernel* size **decreases** incrementally with each layer as a progression (((*convolutional_layers_length* + 1 - *i*) * 2) + 1, ((*convolutional_layers_length* + 1 - *i*) * 2) + 1), e.g. *(7, 7), (5, 5), (3, 3), ...* and *filter* size **decreases** as a progression *np.power(2, 8 - i)*, e.g. *128*, *64*, *32*, ..., where *i* is again the current convolutional layer index.
- Further to this, we provide the option for *kernel* size to remain constant at (*3, 3*) and *filter* size to remain constant at *64*.

For each layer, the non-linearity applied is *ReLU* which has been shown to work well for image classification tasks.

Further to this, *pooling operations* (either *max* / *average* pooling) were applied on top of each *convolutional layer. Pooling* provides a way to summarize feature maps. There is debate [3, 4] on which operation to employ as empirical findings postulate that the operator selector is input domain specific. For example, using the *MNIST* dataset as input benefits from choosing a min pooling operation, as the background of images in the dataset is white (max pixel value is *255*) and the actual feature values (digits) are darker (min pixel value is 0). For our *CNN*, we parameterised the type of pooling operation to be employed as well as kept the parameters *pool size* to *(2, 2)*, strides to *(1, 1)* and padding to be "*same*". Due to lack of a *min pooling* layer in *Keras* we were unable to test with a min pooling strategy. However, we found through experimentation that *average pooling* works best for the task.

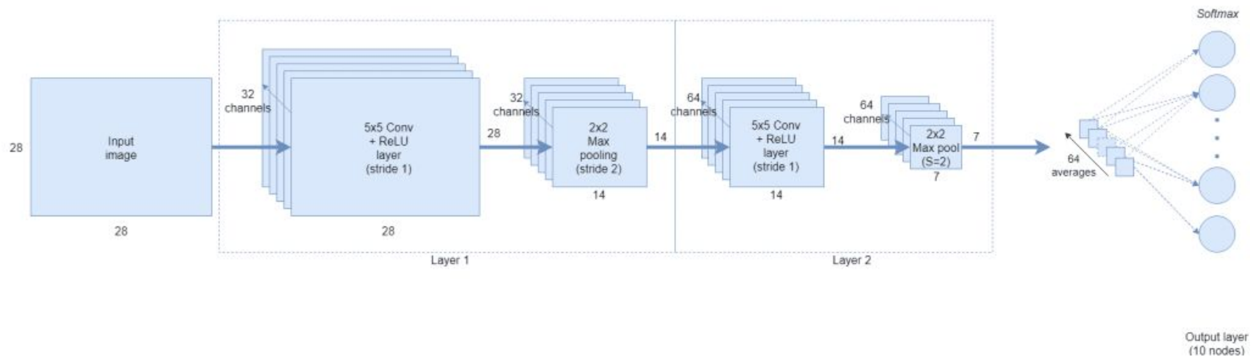


Fig.: Application of *global average pooling* [5]

Next, we applied two forms of *dropout*: *spatial dropout* after each *convolutional layer* and a *dropout layer* on the parameters of the fully connected layer, both with a *rate* of *0.5*. We applied *dropout* so as to avoid overfitting to the training data. For the same reason, we applied a *L2* penalty on the *kernel* and *activation parameters* of the *dense layer* by utilizing functions parameters *kernel_regularizer = tf.keras.regularizers.l2(0.01)* and *activity_regularizer = tf.keras.regularizers.l2(0.01)*, respectively.

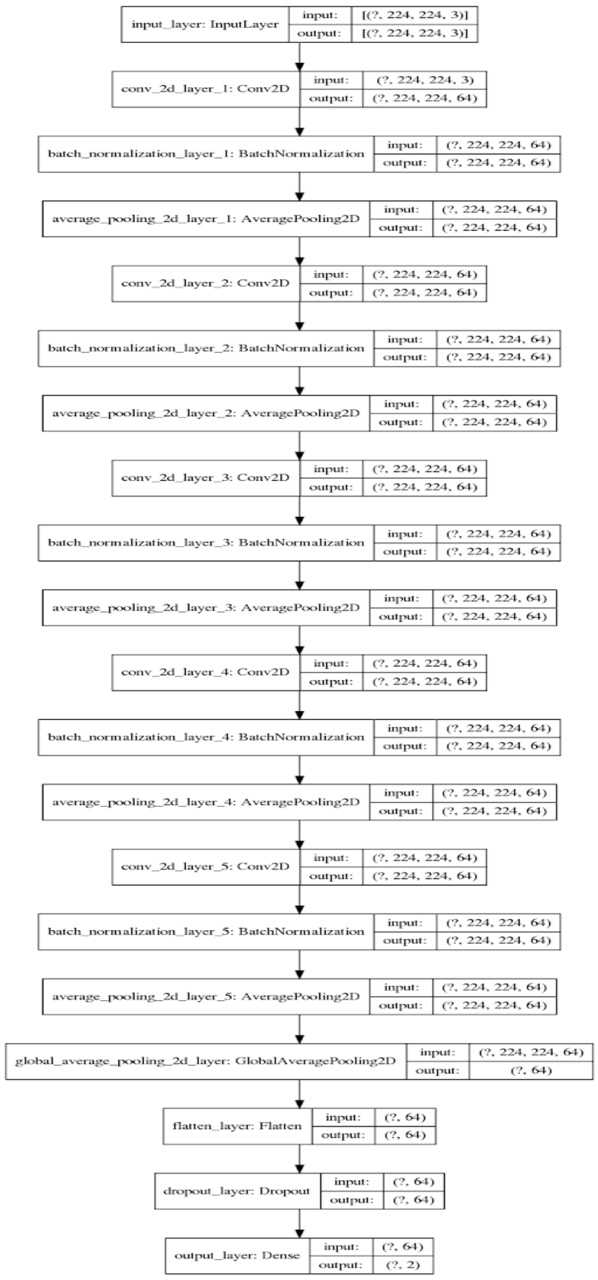
Finally, the output layer of the *CNN* is *2*-unit with a *softmax* function.

[3] "A Gentle Introduction to Pooling Layers for Convolutional ...." 22 Apr. 2019, https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/. Accessed 30 Jun. 2020.

[4] "Maxpooling vs minpooling vs average pooling - Madhushree ...." 7 Feb. 2019, https://medium.com/@bdhuma/which-pooling-method-is-better-maxpooling-vs-minpooling-vs-average-pooling-95fb03f45a9. Accessed 30 Jun. 2020.

[5] "An introduction to Global Average Pooling in convolutional ...." 23 May. 2019, https://adventuresinmachinelearning.com/global-average-pooling-convolutional-neural-networks/. Accessed 30 Jun. 2020.

After heavy manual an iterative experimentation, we concluded that the best architecture, based on the ROC-AUC and Kappa metrics on the test set and also on its balance between performance and training time, was the one of model "*CNN_da_0_ts_224_224_t_0_s_1_bs_8_cl_5_kg_funnel_in_fg_constant_lp_avg_gp_avg_sd_05_fcd_05_all_classes_binary*" which is depicted below:


input_layer: InputLayer | input: [(?, 224, 224, 3)] | output: [(?, 224, 224, 3)]
conv_2d_layer_1: Conv2D | input: (?, 224, 224, 3) | output: (?, 224, 224, 64)
batch_normalization_layer_1: BatchNormalization | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
average_pooling_2d_layer_1: AveragePooling2D | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
conv_2d_layer_2: Conv2D | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
batch_normalization_layer_2: BatchNormalization | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
average_pooling_2d_layer_2: AveragePooling2D | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
conv_2d_layer_3: Conv2D | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
batch_normalization_layer_3: BatchNormalization | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
average_pooling_2d_layer_3: AveragePooling2D | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
conv_2d_layer_4: Conv2D | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
batch_normalization_layer_4: BatchNormalization | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
average_pooling_2d_layer_4: AveragePooling2D | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
conv_2d_layer_5: Conv2D | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
batch_normalization_layer_5: BatchNormalization | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
average_pooling_2d_layer_5: AveragePooling2D | input: (?, 224, 224, 64) | output: (?, 224, 224, 64)
global_average_pooling_2d_layer: GlobalAveragePooling2D | input: (?, 224, 224, 64) | output: (?, 64)
flatten_layer: Flatten | input: (?, 64) | output: (?, 64)
dropout_layer: Dropout | input: (?, 64) | output: (?, 64)
output_layer: Dense | input: (?, 64) | output: (?, 2)

Effectively, our *CNN* has *5 convolutional layers* along with all the other layers described in the general architectural outlook previous. Note that we also tried *3* and *7 convolutional layers*, the former achieving worse results, the latter achieving only marginally better results while also taking a great amount of resources to train.

For reference, this architecture achieved the following performance:

model_id	**test_loss**	**test_accuracy**	**test_roc_auc**
CNN_da_0_ts_224_224_t_0_s_1_bs_8_cl_5_kg_funnel_in_fg_constant_lp_avg_gp_avg_sd_05_fcd_05_all_classes_binary	0.686	0.563	0.606

model_id	**accuracy**	**precision**	**recall**	**f1_score**	**kappa**
CNN_da_0_ts_224_224_t_0_s_1_bs_8_cl_5_kg_funnel_in_fg_constant_lp_avg_gp_avg_sd_05_fcd_05_all_classes_binary	0.563	0.606	0.575	0.534	0.146

As we will see in the sequel, the performance of this architecture is weak when compared to a *DenseNet*. A further comment is that increasing the number of *convolutional layers* does not seem to provide the gains expected, probably due to a vanishing gradients problem which the *DenseNet* architecture alleviates by employing redundant connections between *convolutional layers*, thus facilitating gradient propagation and allowing for greater depth.

As a final note, implementation of the code for building, compiling and fitting our CNN was done in Python class “*Models.CNN*”.

Transfer learning

Choosing the right architecture

DenseNet with pre-trained images on ImageNet

We experimented a with a couple of classification heads on top of the frozen weights of *DenseNet*, specifically,

- A moderately deep *MLP*, comprising *2* fully connected hidden layers of size *1024* and *512* respectively, with dropout layers in between and a *softmax* output layer.

- A single *softmax* activation layer.

Early experiments showed that the latter achieved better performance, so we devoted our resources and effort to training the latter. The entire network architecture is too deep to be depicted in this document, therefore an abstraction of it is presented below, both for the first and second flavors of classification head:

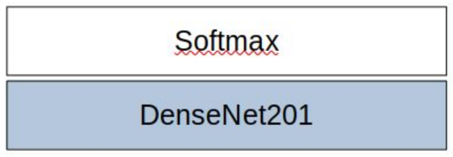


Fig.: A single *softmax* classification head for *DenseNet*.

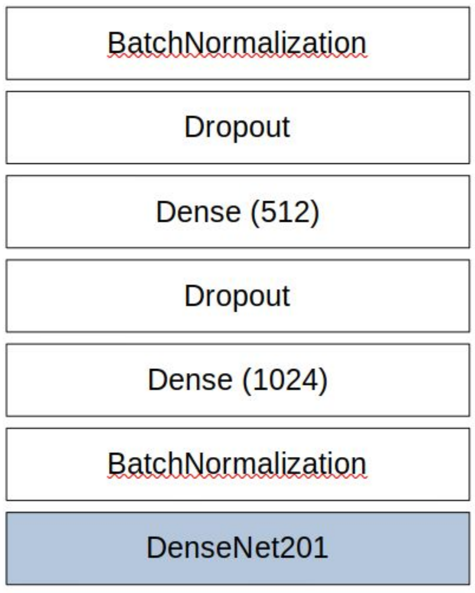


Fig.: A moderately deep *MLP* classification head for *DenseNet*.

Note that we implemented code so as to avoid re-training all *DenseNet* layers, which were kept "frozen". Training was performed only on the classification head that we chose.

The results of our best training / evaluation iteration are depicted below:

model_id	**test_loss**	**test_accuracy**	**test_auc**
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.547	0.724	0.773

model_id	**accuracy**	**precision**	**recall**	**f1_score**	**kappa**
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.724	0.734	0.719	0.718	0.443

It is evident that this architecture is vastly superior to our custom-built *CNN*, as it exploits transfer learning on training performed on *ImageNet*.

Ensemble learning

Ensemble learning is a process by which models trained on a subset of the problem space are combined in order to strengthen their predictive power. Essentially, one employs the "*divide-and-conquer*" technique on the problems space, picking classifiers that behave best on the subset of data, according to their selected metric, and applying cooperative schemes on the set of acquired models for making predictions. The flavors of such schemes, such as being more appropriate for the task at hand, include the following:

Mixture of experts: a set of classifiers trained on specific classes is (e.g. *XR_HAND*, *XR_WRIST* etc.) are linearly combined to produce the output predicted class, effectively giving an average of the groups of model's predictions.

Voting: the majority output class of each model is selected as the actual prediction.

Stacking: stacking works by layering models in a way like collecting the features from models and feeding them into other models to get better results.

Unfortunately, we could not complete the implementation of this part due to lack of time.

Data augmentation

Data augmentation involves the "*artificial variations* (of input data) *to mimic the appearance of future test samples that deviate from the training manifold*" [6].

A very nice feature of *TensorFlow*'s *ImageDataGenerator* is the ability to create altered versions of the input images during model training. Exploiting this capability, we chose to create random augmentations of the input dataset so as to investigate potential performance enhancements in our best model.

For this task, we chose to perform *data augmentation* on our best model only and verify whether this technique would augment our model's predictive capabilities. We chose to randomly vary the data augmentation parameters in the set (*rotation*: *0-30* degrees (parameter chosen from [7]), *flip horizontally*: *True / False*, *flip vertically*: *True / False*).

Although during training the model achieved a validation *ROC-AUC* score of *0.78* on the *9th* epoch of training, the model did not perform better than the highest scopting model on the test set, achieving the results shown below:

model_id	**test_loss**	**test_accuracy**	**test_auc**
DenseNet201_da_1_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.611	0.703	0.748

[6] "Quantifying the effects of data augmentation and stain color ...." 18 Feb. 2019, https://arxiv.org/abs/1902.06543. Accessed 6 Jul. 2020.
[7] "The Effectiveness of Data Augmentation for Detection of ...." 11 Dec. 2017, https://arxiv.org/abs/1712.03689. Accessed 6 Jul. 2020.

model_id	accuracy	precision	recall	f1_score	kappa
DenseNet201_da_1_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.73	0.742	0.694	0.684	0.395

The results are lagging compared to the best results produced by the same architecture without data augmentation, which are shown below:

model_id	test_loss	test_accuracy	test_auc
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.547	0.725	0.773

model_id	accuracy	precision	recall	f1_score	kappa
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.725	0.735	0.720	0.718	0.443

The results are contradicting the findings of research presented in [6] and [7], thereby we conclude that we should perhaps increase the range of parameters used in data augmentation and / or try alternative classification heads.

Experimental results

Evaluation metric (ordered by *test_auc*)

We provide only the top-10 evaluation results for the economy of space, all results can be found in file "*CNN_DenseNet_evaluation_report.csv*".

model_id	test_loss	test_accuracy	test_auc
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.547	0.725	0.773
DenseNet169_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.548	0.722	0.772
DenseNet121_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.559	0.720	0.764

DenseNet169_da_0_ts_224_224_t_0_s_1_bs_4_XR_WRIST_binary	0.626	0.690	0.742
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_4_XR_WRIST_binary	0.664	0.686	0.731
DenseNet121_da_0_ts_224_224_t_0_s_1_bs_4_XR_WRIST_binary	0.681	0.670	0.718
DenseNet169_da_0_ts_224_224_t_0_s_1_bs_4_XR_ELBOW_binary	0.640	0.665	0.703
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_16_XR_FOREARM_binary	0.738	0.641	0.701
DenseNet121_da_0_ts_224_224_t_0_s_1_bs_32_XR_WRIST_binary	1.105	0.654	0.696
DenseNet121_da_0_ts_224_224_t_0_s_1_bs_4_XR_ELBOW_binary	0.687	0.662	0.687

Prediction metrics (ordered by *kappa*)

We provide only the top-10 prediction results for the economy of space, all results can be found in file "*CNN_DenseNet_predictions_report.csv*".

model_id	**accuracy**	**precision**	**recall**	**f1_score**	**kappa**
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.725	0.735	0.720	0.718	0.443
DenseNet169_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.722	0.743	0.715	0.711	0.436
DenseNet121_da_0_ts_224_224_t_0_s_1_bs_4_all_classes_binary	0.720	0.729	0.715	0.714	0.434
DenseNet169_da_0_ts_224_224_t_0_s_1_bs_4_XR_WRIST_binary	0.690	0.700	0.684	0.681	0.372
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_4_XR_WRIST_binary	0.686	0.693	0.681	0.679	0.365
DenseNet121_da_0_ts_224_224_t_0_s_1_bs_4_XR_WRIST_binary	0.670	0.671	0.666	0.666	0.335
DenseNet169_da_0_ts_224_224_t_0_s_1_bs_4_XR_ELBOW_binary	0.665	0.667	0.661	0.660	0.324
DenseNet121_da_0_ts_224_224_t_0_s_1_bs_4_XR_ELBOW_binary	0.662	0.662	0.660	0.660	0.320
DenseNet121_da_0_ts_224_224_t_0_s_1_bs_32_XR_WRIST_binary	0.654	0.657	0.649	0.647	0.301
DenseNet201_da_0_ts_224_224_t_0_s_1_bs_4_XR_FOREARM_binary	0.642	0.646	0.645	0.642	0.287

Please note that the entire set of experiment outputs is provided in the repo of the assignment under the following structure:

- *model/architecture_plots*: graphical output in .png format of all architectures on which experiments were performed.
- *model/checkpoints*: contains the frozen model checkpoint, acquired during the best *early stopping* epoch.
- *model/logs*: contains all training logs of metrics per epoch.
- *report/images*: contains all loss / val_loss and accuracy / val_accuracy training graphs.
- *report/summaries*: contains all .csv files with summary performance data for each architecture (validation and predictions report).

Conclusions

Transfer learning took the world of ML by storm a few years ago by allowing the re-use of feature weights of pre-trained models, trained in large scale and on massive datasets, to be used effectively for downstream ML tasks. Our own research attests to the vast improvement of key metrics when transfer learning is employed, specifically the use of *DenseNet* architecture for the image classification task.